Índice (I)

Marina de la Cruz
Alfonso Ortega

Índice (II)

Descripción de la calculadora

- El objetivo es **construir una calculadora utilizando las herramientas Flex y Bison** que cumpla las siguientes especificaciones.
- La calculadora tiene un funcionamiento similar al de un **intérprete**, cuando **recibe como entrada una expresión matemática** como por ejemplo 6+3, **calcula el resultado de la operación y lo muestra por pantalla**.
- Los tipos de datos que puede manipular la calculadora son **entero y real**.
  - **El formato de los números enteros** es el habitual, es decir, un número entero es una secuencia de uno o más dígitos del 0 al 9.
  - **El formato de los números reales** consiste en un parte entera formada por una secuencia de cero o más dígitos del 0 al 9, a continuación un punto, y seguidamente la parte fraccionaria formada por una secuencia de uno o más dígitos del 0 al 9.
- **Las operaciones disponibles** son: suma, resta, multiplicación y división para el tipo de datos real, y suma, resta y multiplicación para el tipo de datos entero. No está permitida ninguna operación aritmética con operandos de tipos diferentes.
- Todas las operaciones se pueden agrupar con paréntesis.
- Una expresión termina con un salto de línea.
- El usuario puede introducir espacios en blanco y tabuladores, y todos serán ignorados. Las líneas vacías no están permitidas (una línea en blanco es un error sintáctico)

Reglas de la gramática

- El lenguaje que entiende la calculadora es el que genera la siguiente gramática, **excluyendo las reglas que describen las constantes enteras y las reales**:

```
1:    <linea>          ::=    <exp> \n
2:    <exp>            ::=    <exp real>
3:                      |     <exp entera>
4:    <exp entera>     ::=     entero
5:                      |     <exp entera> + <exp entera>
6:                      |     <exp entera> - <exp entera>
7:                      |     <exp entera> * <exp entera>
8:                      |     ( <exp entera> )
9:    <exp real>       ::=     real
10:                     |     <exp real > + <exp real >
11:                     |     <exp real > - <exp real >
12:                     |     <exp real > * <exp real >
13:                     |     <exp real > / <exp real >
14:                     |     ( <exp real > )
```

Los tokens y sus expresiones regulares para Flex

Ejercicio

- Identificar los tokens del lenguaje de la calculadora y definir sus correspondientes expresiones regulares para Flex.

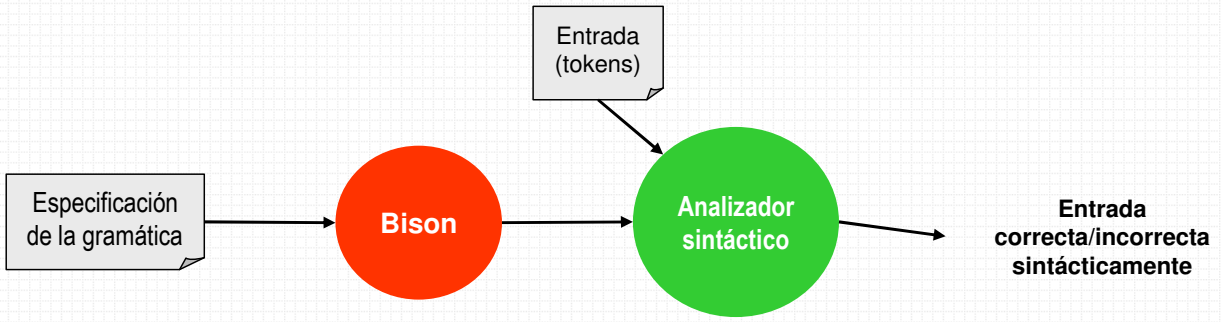
Los tokens y sus expresiones regulares para Flex

Solución

- Identificar los tokens del lenguaje de la calculadora y definir sus correspondientes expresiones regulares para Flex.
- Operadores: “+” “-” “*” “/”
- Símbolos: “(” “)” “\n”
- Números enteros: [0-9]+
- Números reales: [0-9]*”.”[0-9]+
- Caracteres a ignorar: “ ” “ \t”

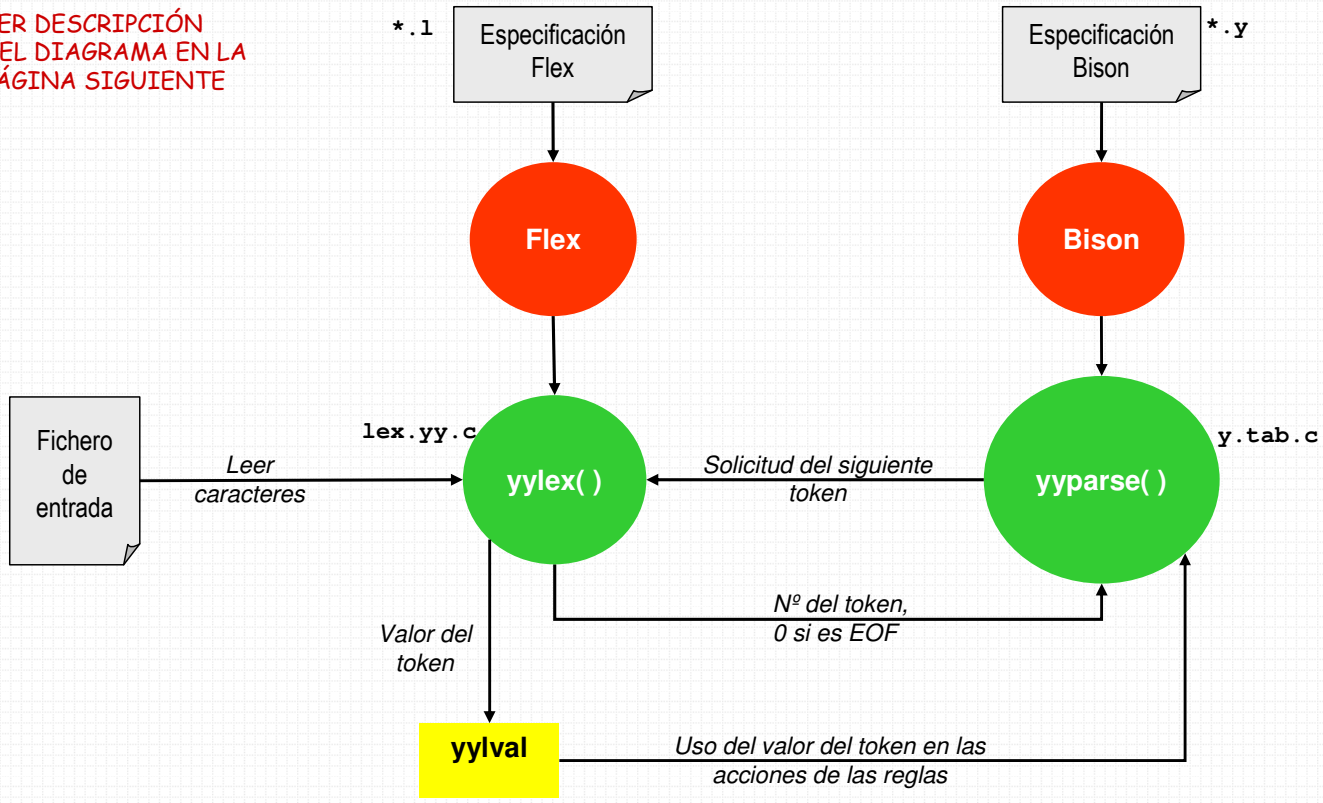
Cómo trabajan Flex y Bison (I)

- Flex *versus* Bison
  - Flex reconoce expresiones regulares, y Bison reconoce gramáticas.
  - Flex divide el programa fuente en tokens y Bison agrupa esos tokens.
- Es habitual utilizar Flex y Bison conjuntamente para desarrollar un compilador.
  - Con Flex se construye el analizador léxico (morfológico)
  - Con Bison se desarrolla el analizador sintáctico.
- Bison construye, a partir de la gramática especificada por el usuario, un analizador sintáctico que reconoce entradas sintácticamente correctas para dicha gramática.



Cómo trabajan Flex y Bison (II)

VER DESCRIPCIÓN DEL DIAGRAMA EN LA PÁGINA SIGUIENTE



Cómo trabajan Flex y Bison (III)

- **Flex** construye la función **yylex()** de **análisis morfológico** a partir del fichero de especificación correspondiente. La función **yylex()** lee el fichero de entrada e identifica los tokens.

- **Bison** construye la función **yyparse()** de **análisis sintáctico** a partir del fichero de especificación correspondiente. La función **yyparse()** solicita a la función **yylex()** los tokens de la entrada y comprueba si forman una construcción válida de acuerdo a las reglas de la gramática descritas en el fichero de especificación.

- Cada vez que la función **yylex()** devuelve un token al analizador sintáctico, si el token tiene un valor asociado, éste se guarda en la variable **yylval** antes de terminar. Por ejemplo, un identificador de una variable, además de tener un número de token que lo identifica y distingue de otros tipos de tokens, también tiene asociado como valor o atributo, el lexema del identificador. Sin embargo, un paréntesis no tiene asociado ningún valor o atributo. Posteriormente se estudiará cómo la función **yyparse()** utiliza los valores asociados a los tokens y cómo definir el tipo de la variable **yylval** (por defecto es de tipo int).

Breve descripción del fichero de especificación de Bison (I)

- El fichero de especificación de Bison tiene una estructura similar al fichero de especificación de Flex. Se compone de tres secciones separadas por líneas que contienen el separador %%.

sección de definiciones

```
%{
    /* delimitadores de código C */
%}

%%
```

sección de reglas

```
%%
```

sección de funciones de usuario

Breve descripción del fichero de especificación de Bison (II)

- **Sección de definiciones**, contiene:
  - La definición de los tokens.
  - La definición de los diferentes tipos de los tokens.
  - La definición del axioma de la gramática.
  - La definición de la precedencia y la asociatividad de los operadores.
  - Código C que se copia literalmente en el fichero de salida y que normalmente contiene declaraciones de variables y funciones que se utilizan en la sección de reglas. El código C va encerrado entre líneas con los caracteres %{ y %}.
- **Sección de reglas**: contiene las reglas de la gramática en un formato concreto.
- **Sección de funciones de usuario**, contiene:
  - Funciones escritas por el usuario para ser utilizadas en la sección de reglas, es decir, funciones de soporte.
  - Se copia literalmente en el fichero de salida.
  - En aplicaciones grandes, es más conveniente agrupar todas las funciones de soporte en un fichero o conjunto de ficheros, en lugar de incluirlas en esta sección.

Especificaciones para Flex y Bison (I)

Definición de los tokens cuando se trabaja sólo con Flex

- Se escribe un fichero de cabecera, por ejemplo, **tokens.h** con la definición de los tokens y se incluye ese fichero en el fichero de especificación de Flex.
- Se debe incluir el fichero **tokens.h** en cualquier fichero que contenga llamadas a la función de análisis morfológico **yylex()**, ya que dicha función devuelve como valor de retorno los tokens definidos en el fichero de cabecera.

tokens.h

```
#ifndef _TOKENS_H
#define _TOKENS_H

#define TOK_REAL 1
#define TOK_ENTERO 2
#define TOK_MAS 3
#define TOK_MENOS 4
#define TOK_POR 5
#define TOK_DIV 6
#define TOK_PARIZQ 7
#define TOK_PARDER 8
#define TOK_SALTO 9

#define TOK_ERROR -1

#endif
```

Especificaciones para Flex y Bison (II)

Definición de los tokens cuando se trabaja con Flex y Bison

- Definición de tokens:
  - No es necesario definr los tokens en un fichero de cabecera.
  - Se definen los tokens utilizando la declaración **%token** en la sección de definiciones del fichero de especificación de Bison.
  - Los tokens de un sólo carácter no es necesario definirlos, ya están predefinidos en Bison por su correspondiente valor ASCII. Por lo tanto, para la calculadora sólo se definen los tokens TOK_REAL y TOK_ENTERO.
- Definición de los tipos de valores de los tokens:
  - Se definen los distintos tipos de los valores de los tokens con la declaración **%union** en la sección de definiciones. A partir de esta declaración Bison crea una definición C de tipo union y de nombre **YYSTYPE**. También define la variable **yylval** de ese tipo.
  - Se añade a cada declaración **%token** el campo del tipo union correspondiente al valor del token.

calc.y

```
%{
.....

%}

%union
{
      double real;
      int entero;
}
%token <real> TOK_REAL
%token <entero> TOK_ENTERO

%%

.....

%%

.....
```

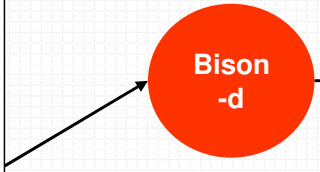

Especificaciones para Flex y Bison (III)

Acceso a la definición de los tokens desde Flex

- Se compila con Bison el fichero calc.y con la **opción -d** para que se genere el fichero **y.tab.h** que contiene la definición de los tokens y sus tipos.
- En el fichero de especificación de Flex se incluye el fichero **y.tab.h**

calc.y

```
%{
.....

%}

%union
{
      double real;
      int entero;
}
%token <real> TOK_REAL
%token <entero> TOK_ENTERO

%%

.....

%%

.....
```



y.tab.h

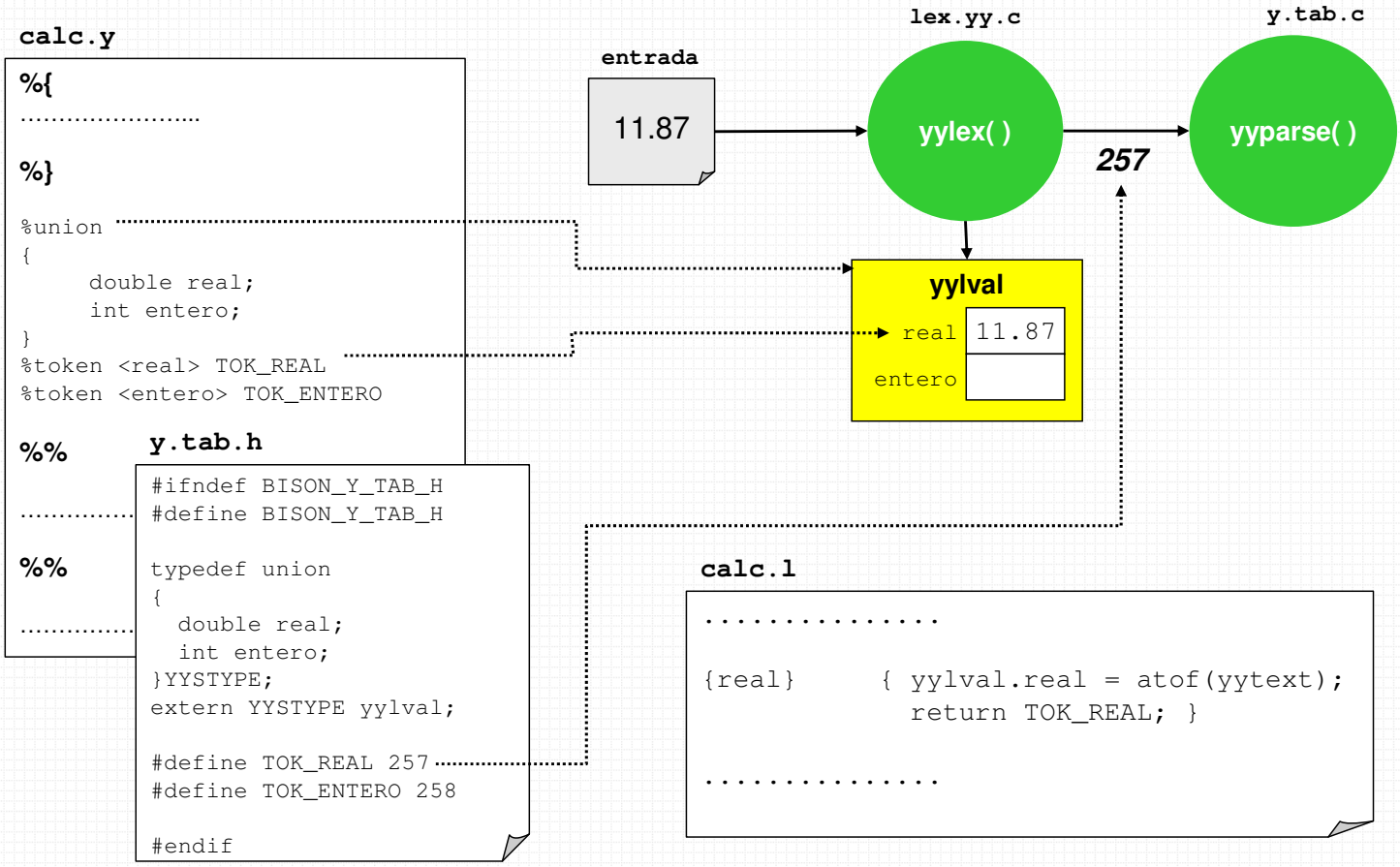
```
#ifndef BISON_Y_TAB_H
#define BISON_Y_TAB_H

typedef union
{
  double real;
  int entero;
}YYSTYPE;
extern YYSTYPE yylval;

#define TOK_REAL 257
#define TOK_ENTERO 258

#endif
```

calc.l

```
%{
#include "y.tab.h"
.....
```

Especificaciones para Flex y Bison (IV)

Especificación para Flex



Especificaciones para Flex y Bison (V)

Ejercicio

- Escribir todas las secciones del fichero de especificación Flex **calc.l**
- No olvidar que los tokens de un solo caracter están predefinidos en Bison por su correspondiente valor ASCII, y por lo tanto, la función de análisis léxico, cuando detecte este tipo de token debe devolver su valor ASCII.

Especificaciones para Flex y Bison (VI)

Solución

calc.l

Includes para usar las funciones `atoi, atof y fprintf`

Fichero de cabecera generado por Bison

Token para errores morfológicos

Opción para evita tener que definir la función `yywrap()`

```
%{
#include <stdlib.h>
#include <stdio.h>
#include "y.tab.h"
#define TOK_ERROR -1
%}
%option noyywrap
blanco   [ \t]
entero   [0-9]+
real     [0-9]*"."[0-9]+
simbolo  "+"|"-"|"*"|"/"|"("|")"|\n

%%
{blanco}  ;
```

Ignorar blancos (tabuladores y espacios en blanco)

Número entero. Se devuelve TOK_ENTERO y su valor a través de `yylval`

```
{entero}    { yylval.entero = atoi(yytext);
              return TOK_ENTERO; }
```

Número entero. Se devuelve TOK_REAL y su valor a través de `yylval`

```
{real}      { yylval.real = atof(yytext);
              return TOK_REAL; }
```

Regla para los operadores aritméticos, los paréntesis y el salto de línea

```
{simbolo}   { return yytext[0]; }
```

Regla para error morfológico

```
.           {fprintf("error morfológico\n");
             return TOK_ERROR;}
%%
```


Especificaciones para Flex y Bison (VII)

Incorporación de las reglas de la gramática (I)

- La sección de reglas dentro del fichero de especificación de Bison es una lista de todas las reglas de la gramática, con un formato determinado.
- El formato de cada regla es muy similar a la notación BNF, con las siguientes diferencias:
  - se separa la parte izquierda de la derecha con el carácter dos puntos.
  - cada regla se termina con un punto y coma.
- En las partes derechas de las reglas, los tokens de un solo carácter se escriben encerrados entre comillas simples.

calc.y

```
%{
.....................
%}
%union
{
     double real;
     int entero;
}
%token <real> TOK_REAL
%token <entero> TOK_ENTERO

%%
linea:    exp '\n' {}
         ;
exp:      exp_real {}
         |exp_entera {}
         ;
```

(continúa ...)

Especificaciones para Flex y Bison (VIII)

Incorporación de las reglas de la gramática (II)

calc.y

(... continuación)

```
exp_entera:        TOK_ENTERO {}
                  |exp_entera '+' exp_entera {}
                  |exp_entera '-' exp_entera {}
                  |exp_entera '*' exp_entera {}
                  |'(' exp_entera ')' {}
                  ;
exp_real:          TOK_REAL {}
                  |exp_real '+' exp_real {}
                  |exp_real '-' exp_real {}
                  |exp_real '*' exp_real {}
                  |exp_real '/' exp_real {}
                  |'(' exp_real ')' {}
                  ;

%%
..............
```


Especificaciones para Flex y Bison (IX)

Especificación del axioma de la gramática y características de los operadores

- Axioma de la gramática
  - Bison permite especificar el axioma de la gramática con la declaración de la forma: **%start <axioma>**
  - Si se omite la declaración, se asume que el axioma de la gramática es el primer no terminal de la sección de reglas de la gramática.
- Características de los operadores
  - Se puede definir la asociatividad de los operadores con las declaraciones **%left** o **%right**.
  - La precedencia de los operadores se determina por el orden de aparición de las declaraciones en la sección de definiciones. Los operadores que aparecen primero tienen menor precedencia.

calc.y

```
%{
.....................
%}
%union
{
      double real;
      int entero;
}
%token <real> TOK_REAL
%token <entero> TOK_ENTERO

%start linea
%left '+' '-'
%left '*' '/'


%%
.....................
%%
```

Especificaciones para Flex y Bison (X)

Incorporación de la funciones yyerror() y main() (I)

- Función **yyerror()**
  - Cuando Bison detecta un error sintáctico, invoca a la función **yyerror()**. Esta función tiene que ser proporcionada por el usuario, y se puede incorporar en la última sección del fichero de especificación calc.y (en Linux se puede no proporcionar ya que la proporciona la librería de Bison).
  - El prototipo de la función es:

    void yyerror(char* s)
  - El único parámetro de la función es el mensaje del error sintáctico ocurrido. Bison habitualmente invoca a esta función con el mensaje "Syntax error" o bien "Parser error", sin más información. Es responsabilidad del usuario de Bison completar este mensaje con información acerca de:
    - El tipo de error ocurrido
    - La ubicación del error dentro del fichero fuente (fila y columna)
  - La versión básica de esta función muestra por pantalla el mensaje que recibe como parámetro.
- Función **main()**
  - La rutina de análisis sintáctico **yyparse()** tiene que ser invocada. Para realizar pruebas, se puede incorporar una función **main()** en la sección de funciones de usuario del fichero de especificación calc.y (en Linux se puede no proporcionar ya que la proporciona la librería de Bison). Cuando se construya el compilador completo, se eliminará esta función **main()** y se invocará a la rutina de análisis sintáctico **yyparse()** desde fuera del fichero de especificación **calc.y**

Especificaciones para Flex y Bison (XI)

Incorporación de la funciones yyerror() y main() (II)

calc.y

```
%%

int main()
{
        return(yyparse());
}

void yyerror(char* s)
{
        fprintf(stderr,"%s\n",s);
}
```

No olvidar incluir en la primera sección del fichero `calc.y` la directiva `#include <stdio.h>`

Creación del ejecutable

- Con el contenido actual de los ficheros de especificación calc.l y calc.y se puede realizar una primera prueba de la calculadora. De momento, la calculadora no realiza las operaciones, pero sí se puede verificar que el análisis léxico funciona correctamente y también el análisis sintáctico.

- Para crear el ejecutable:

  - Compilar la especificación flex:
    flex calc.l se crea el fichero lex.yy.c

  - Compilar la especificación Bison:
    bison -d -y calc.y se crean los ficheros y.tab.h, y.tab.c

  - Generar el ejecutable:
    gcc -Wall -o calc lex.yy.c y.tab.c se crea el fichero calc

Primera prueba de la calculadora (I)

Entradas correctas

- Arrancar la calculadora y:
  - Probar las siguientes expresiones morfológica y sintácticamente correctas.
  - Terminar cada expresión con un salto de línea.
  - Terminar las pruebas con el carácter de finalización de la entrada (CTRL-D)

- Expresiones constantes
  - 12
  - 3.7
  - .9
- Expresiones enteras
  - 4+4
  - 9-5
  - 7*3
  - (4+5) * (9-6)
- Expresiones reales
  - 9.4 + 7.2
  - 10.8 - 2.0
  - .9 * 2.3
  - 90.8 / 3.5
  - (4.9 - 1.2) / (2.0 + 6.5)
- Otros
  - (((6 - 4)))

- Se puede observar que la calculadora acepta como válidas las expresiones anteriores (aunque de momento no devuelve ningún resultado).

Primera prueba de la calculadora (II)

Entradas incorrectas

- Arrancar la calculadora y:
  - Probar las siguientes expresiones incorrectas y observar la diferencia de los mensajes de error en cada caso.
  - Terminar cada expresión con un salto de línea.
  - Terminar las pruebas con el carácter de finalización de la entrada (CTRL-D)
  - 3.
    - Detrás del punto decimal tiene que aparecer al menos una cifra.
    - El mensaje por pantalla es “error morfológico” y “parser error”/“syntax error”. Ha ocurrido un error morfológico, y eso ha generado un error sintáctico.
  - 3,7
    - El separador decimal no es una coma.
    - El mensaje por pantalla es “error morfológico” y “parser error”/“syntax error”. Ha ocurrido un error morfológico, y eso ha generado un error sintáctico.
  - 9/8
    - La operación de división no está permitida entre números enteros.
    - El mensaje por pantalla es “parser error”/“syntax error”. No hay error morfológico
  - 2.3*6
    - Las operaciones entre números de distinto tipo no están permitidas.
    - El mensaje por pantalla es “parser error”/“syntax error”. No hay error morfológico.

Funcionamiento de la función **yyparse()** (I)

Funcionamiento básico

- El analizador sintáctico busca reglas que concuerden con los tokens de la entrada.

- El analizador sintáctico generado por Bison es de tipo **LALR(1)**.
  - Los **estados** del autómata están formados por posibles posiciones en una o más reglas parcialmente analizadas.
  - Cada vez que se lee un nuevo token, si no se puede completar una regla, el analizador apila el token en una pila interna y transita a un nuevo estado. A esta acción se la denomina **desplazamiento** (shift).
  - Cuando se lee un nuevo token, que permite completar una regla, es decir, que es el último que faltaba de la parte derecha de una regla, el analizador desapila todos estos símbolos de la pila interna, apila el símbolo de la parte izquierda de la regla y transita a un nuevo estado. A esta acción se la denomina **reducción** (reduction).
  - **Cada vez que se reduce una regla, se ejecuta el código que tiene asociado**, que habitualmente se denomina acción semántica. Este código lo aporta el usuario de Bison. Este es el mecanismo que permite **añadir funcionalidad al análisis sintáctico**, es decir, no sólo analizar sintácticamente una entrada sino hacer algo más, generar alguna salida. Esa salida se genera en la ejecución del código asociado a las reglas. Por ejemplo, cuando se desarrolla un compilador, el código asociado a las reglas se puede utilizar para realizar el análisis semántico y generar código.

Funcionamiento de la función **yyparse()** (II)

Los valores de los símbolos (I)

- En la pila interna del analizador sintáctico, además de apilarse los símbolos, también se apilan sus correspondientes valores semánticos o atributos (para los tokens, intercambiados a través de la variable **yylval**)
- Desde el código asociado a una regla se puede **acceder** a los valores semánticos de los símbolos de la parte derecha de la regla mediante `$1`, `$2`,... , y **establecer** el valor del símbolo de la parte izquierda de la regla, a través de `$$`.
  Por ejemplo, la regla:

```
exp_entera : exp_entera '+' exp_entera
```

  podría tener asociada la siguiente acción:

```
{ $$ = $1 + $3 }
```

  que calcula el valor semántico o atributo del símbolo de la parte izquierda de la regla como la suma de los atributos de los símbolos primero y tercero de la parte derecha de la regla.
- En las reglas que no tienen acción asociada se utiliza la acción por defecto:

```
{ $$ = $1 }
```


Funcionamiento de la función **yyparse()** (III)

Los valores de los símbolos (II)

- Dado que un símbolo no terminal puede tener asociado un valor semántico, es necesario poder definir el tipo de dicho valor, de la misma manera que se define el tipo de los valores semánticos de los símbolos terminales o tokens.
- La definición del tipo de valor de un no terminal se hace en la sección de definiciones del fichero de especificación de Bison con la declaración **%type**, que es similar a la declaración **%token** que se utiliza para definir el tipo de dato de los tokens.
- En la declaración **%type** se especifica el nombre del campo de la estructura union que corresponde al tipo de dato del no terminal.
- Después de las declaraciones **%token** y **%type**, cualquier referencia a un valor a través de $$, $1, etc, accede al campo concreto dentro de la estructura union.
- Por ejemplo, en la calculadora, para definir como entero el tipo de valor semántico del símbolo no terminal exp_entera, se hace:

```
%type <entero> exp_entera
```

Recordar que la definición de la estructura union es:

```
%union {
        double real;
        int entero;  }
```

Incorporación de funcionalidad a la calculadora (I)

- **Mediante las acciones asociadas a las reglas y el cálculo de los valores `$$` a partir de los valores `$1`, `$2`**, . . . se incorpora a la calculadora la funcionalidad de realizar la operación introducida por el usuario y mostrar el resultado por pantalla.

- Antes de añadir las acciones a las reglas, se define el tipo de valor semántico de los símbolos no terminales que tengan valor semántico. En el caso de la calculadora, para conseguir la funcionalidad descrita en sus requisitos, sólo es necesario definir el tipo de valor de los símbolos no terminales `exp_entera` y `exp_real`. Para ello, en la sección de definiciones se añade:

```
%type <real> exp_real
%type <entero> exp_entera
```

Recordar que la definición de la estructura union es:

```
%union {
        double real;
        int entero;  }
```

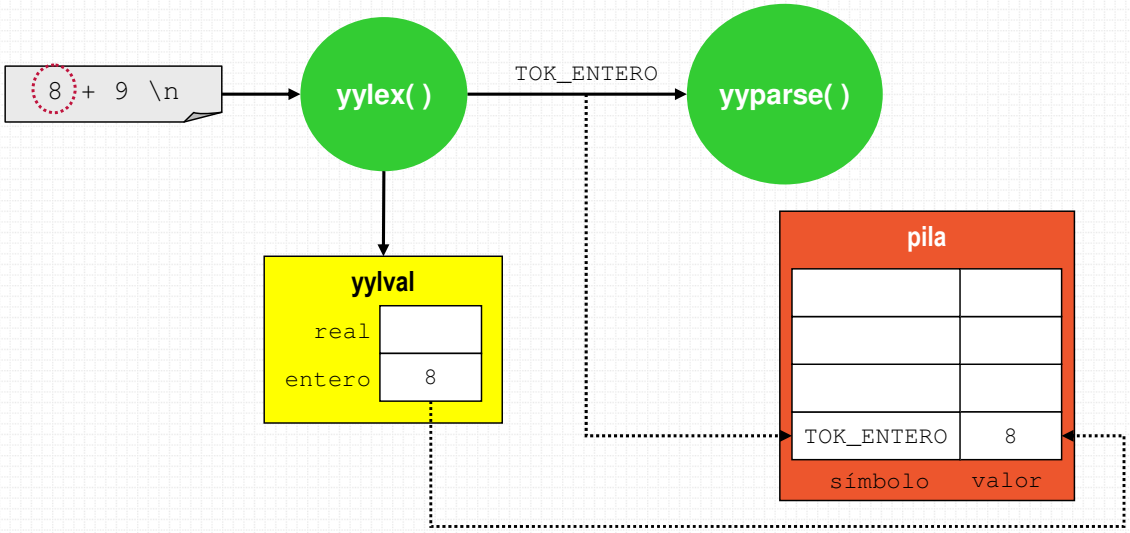

Incorporación de funcionalidad a la calculadora (II)

En el fichero **calc.y** se añaden las siguientes acciones a las reglas:

```
linea:          exp '\n' {}
                ;
exp:             exp_real        {printf("%f\n", $1);}
                |exp_entera      {printf("%d\n", $1);}
                ;
exp_entera:      TOK_ENTERO                          { $$ = $1;}
                |exp_entera '+' exp_entera           { $$ = $1 + $3;}
                |exp_entera '-' exp_entera           { $$ = $1 - $3;}
                |exp_entera '*' exp_entera           { $$ = $1 * $3;}
                |'(' exp_entera ')'                  { $$ = $2;}
                ;
exp_real:        TOK_REAL                            { $$ = $1;}
                |exp_real '+' exp_real               { $$ = $1 + $3;}
                |exp_real '-' exp_real               { $$ = $1 - $3;}
                |exp_real '*' exp_real               { $$ = $1 * $3;}
                |exp_real '/' exp_real               { if ($3) $$=$1/$3;
                                                       else $$=$1; }
                |'(' exp_real ')'                    { $$ = $2;}
                ;
```

Análisis de la entrada "8+9 \n" (I)

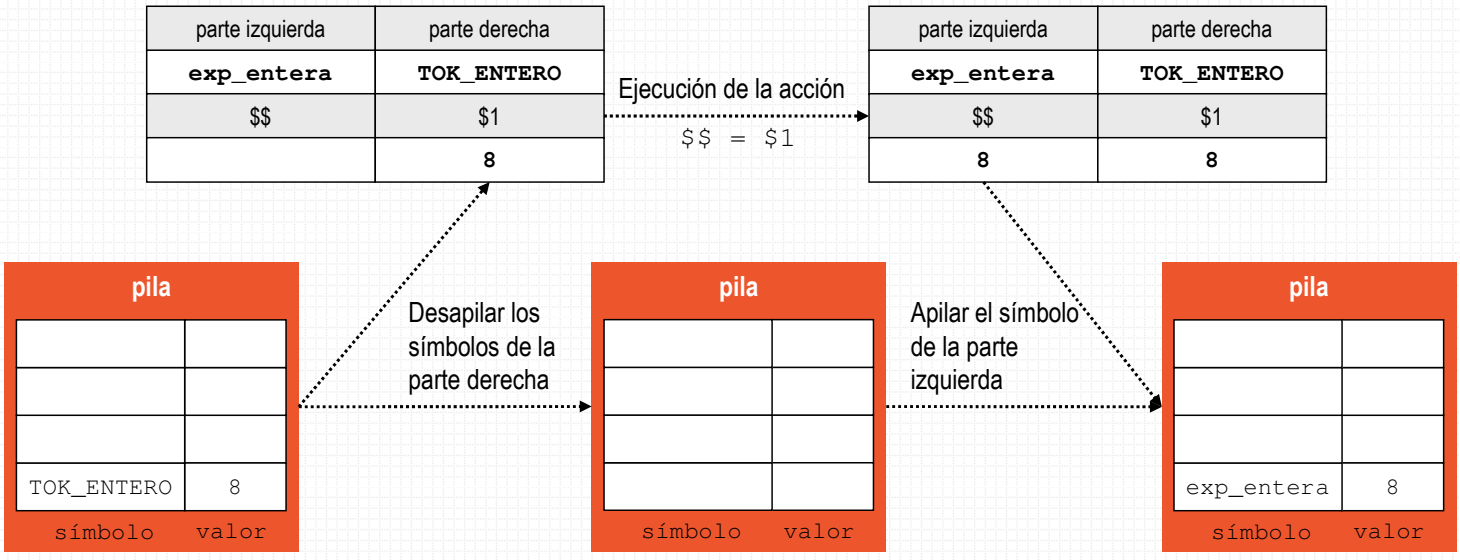
- En las páginas siguientes se muestra la secuencia de operaciones que realiza la función **yyparse()** durante el análisis de la entrada "8 + 9 \n". Son operaciones de reducción, de desplazamiento y de ejecución de las acciones asociadas a las reglas. También se incluyen las operaciones que realiza **yylex()**. Por claridad, en la pila de análisis sólo se muestran los símbolos y se han omitido los estados.



`yylex()` lee de la entrada el `8`, y pasa la información a `yyparse()` mediante el valor de retorno de `yylex()` y la variable `yylval`.

`yyparse()`, realiza una operación de desplazamiento apilando el símbolo y su correspondiente valor.

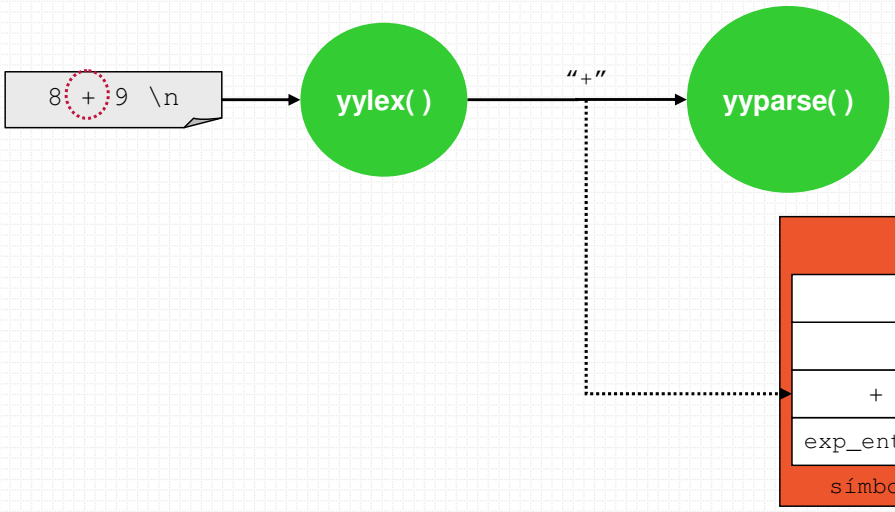
Análisis de la entrada "8+9 \n" (II)



`yyparse()` realiza una operación de reducción de la regla `exp_entera:TOK_ENTERO`

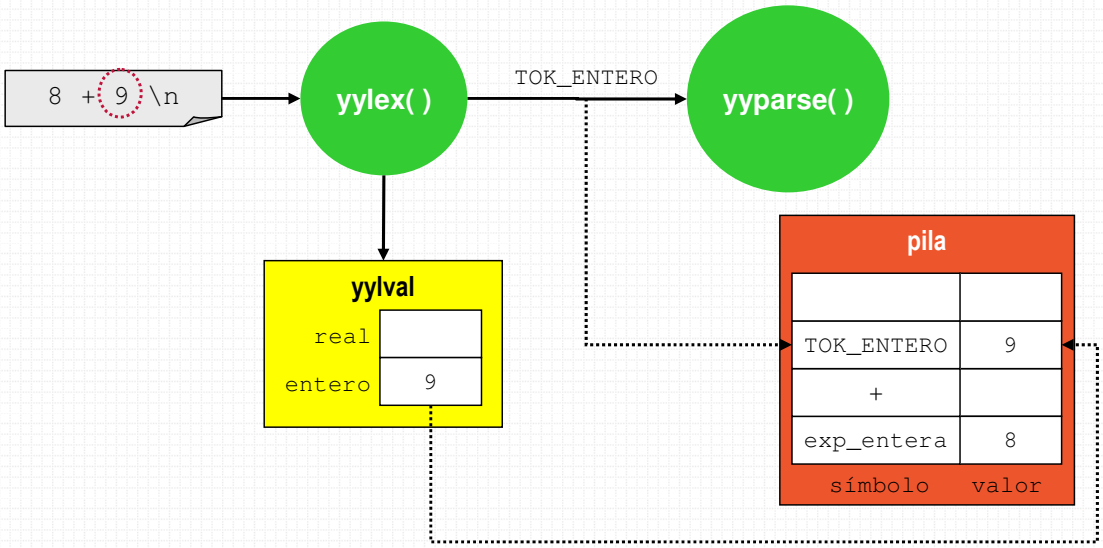
- se desapilan los símbolos de la parte derecha de la regla junto con sus valores semánticos
- se ejecuta la acción asociada a la regla `{$$=$1}`. En este punto se dispone de `$1`.
- se apila el símbolo de la parte izquierda junto con su valor semántico

Análisis de la entrada "8+9 \n" (III)



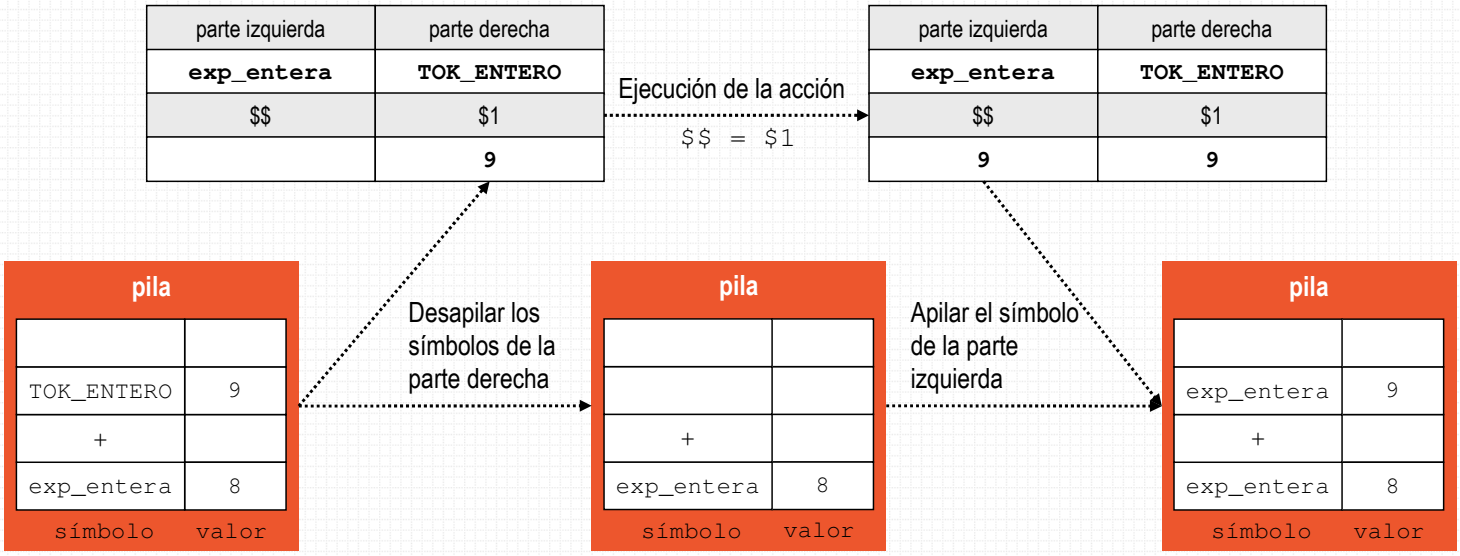
`yylex()` lee de la entrada el `+`, y pasa la información a `yyparse()` mediante el valor de retorno de `yylex()`.	`yyparse()`, realiza una operación de desplazamiento apilando el símbolo (en este caso el símbolo "+" no tiene asociado valor semántico).

Análisis de la entrada "8+9 \n" (IV)



`yylex()` lee de la entrada el `9`, y pasa la información a `yyparse()` mediante el valor de retorno de `yylex()` y la variable `yylval`.	`yyparse()`, realiza una operación de desplazamiento apilando el símbolo y su correspondiente valor.

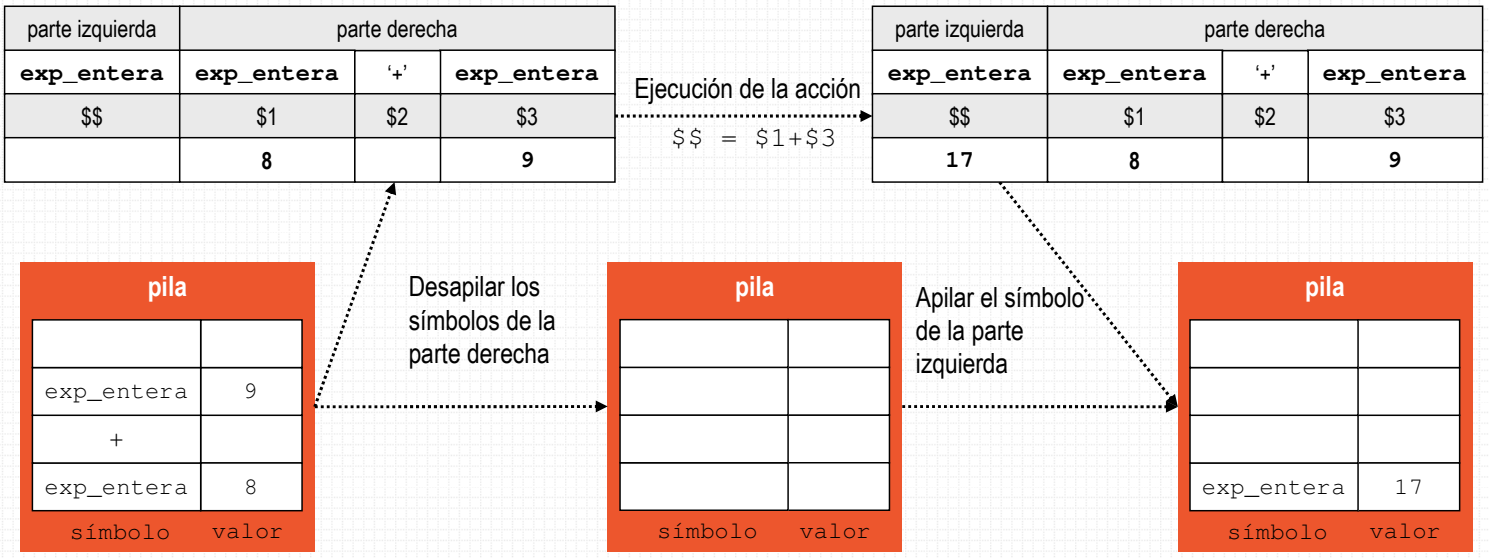
Análisis de la entrada "8+9 \n" (V)



`yyparse()` realiza una operación de reducción de la regla `exp_entera:TOK_ENTERO`

- se desapilan los símbolos de la parte derecha de la regla junto con sus valores semánticos
- se ejecuta la acción asociada a la regla `{$$=$1}`. En este punto se dispone de `$1`.
- se apila el símbolo de la parte izquierda junto con su valor semántico

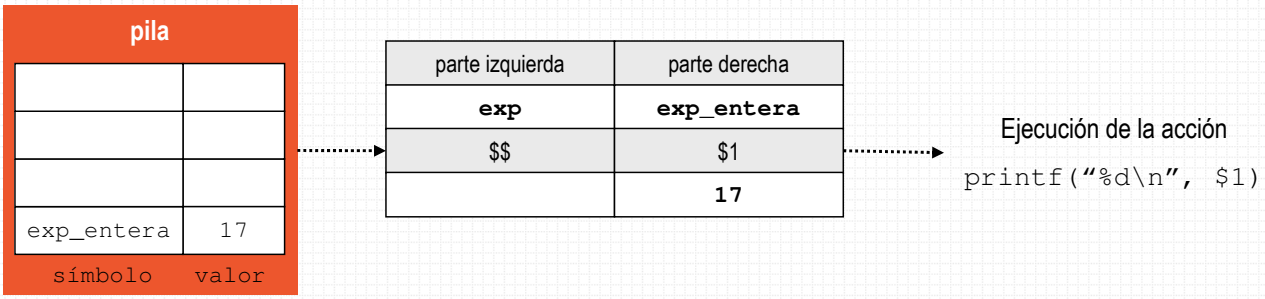
Análisis de la entrada "8+9 \n" (VI)



`yyparse()` realiza una operación de reducción de la regla `exp_entera:exp_entera '+' exp_entera`

- se desapilan los símbolos de la parte derecha de la regla junto con sus valores semánticos
- se ejecuta la acción asociada a la regla `{$$=$1+$3}`. En este punto se dispone de `$1` y `$3`.
- se apila el símbolo de la parte izquierda junto con su valor semántico

Análisis de la entrada "8+9 \n" (VII)

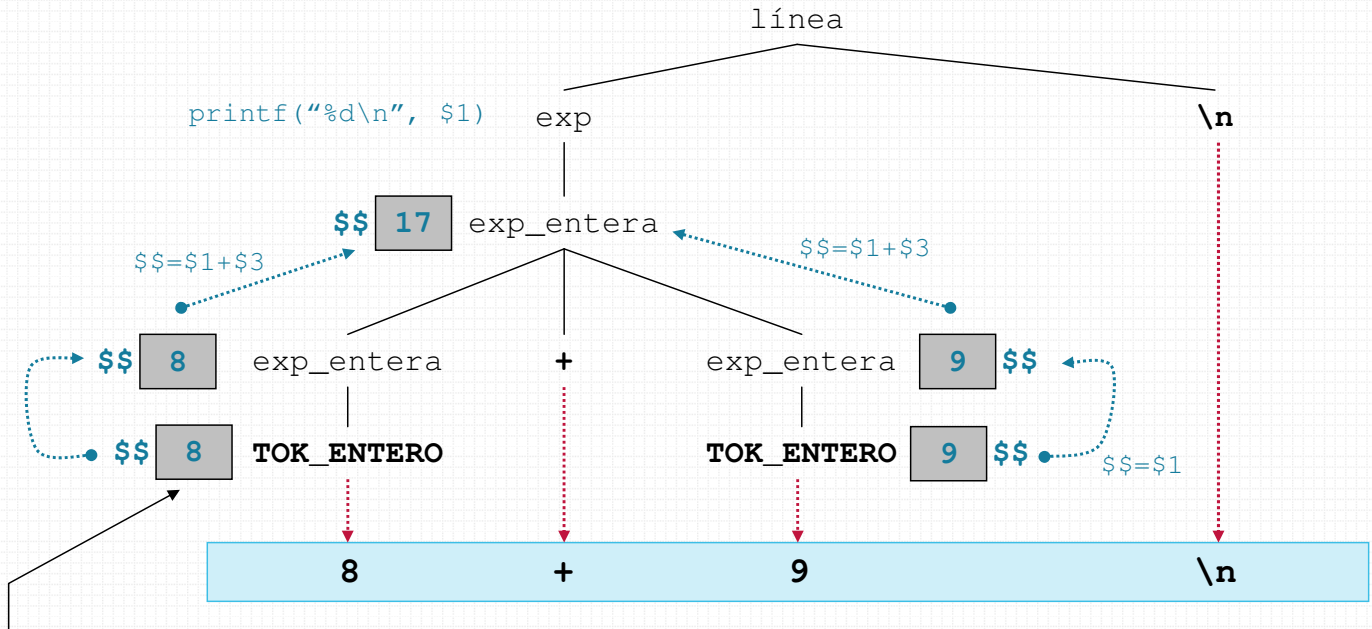


`yyparse()` realiza una operación de reducción de la regla `exp : exp_entera`

- se desapilan los símbolos de la parte derecha de la regla junto con sus valores semánticos
- se ejecuta la acción asociada a la regla `printf("%d\n", $1)`. En este punto se dispone de `$1`.

La propagación de atributos vista sobre el árbol de análisis

- El árbol de análisis sintáctico permite visualizar el cálculo de atributos sintetizados en las acciones asociadas a las reglas. Se observa claramente la propagación de los atributos por los nodos del árbol.



El analizador léxico actualiza la variable `yylval` con el valor semántico del último token reconocido.